CSC A48 Winter 2019 Term Test Study Guide

Section 0: Introduction to C and Basic Data Types

C is an **imperative** language. There is no interactive mode in C. All programs start at a function called `main()`. Program can include and use code from **libraries**, similar to Python modules. Here is an example of the most famous program in C.

```c
// Allows you to use input and output command such as printf
#include <stdio.h>

int main()
{
  // end your print statement with a newline character
  printf("HELLO WORLD\n");
  // your main function must return 0 if successful
  return 0;
}
```

The fundamental data types supported by C are: 1. `int` which is an integer number, 2. `float/double`, a floating point number 3. `char` which is one character and 4. `void` which represents no data type attached like `None` in Python.

```c
#include <stdio.h>

int main()
{
  float my_float = 10.0;
  int my_int = 3;
  char my_char 'a';
  printf("My variables are: %f, %d, %c \n", my_float, my_int, my_char);
  return 0;
}
```

String Operations

Strings in C

Recall that strings in C are not primitive. Strings are an **array of characters**, which is terminated with a **null character** ∼~~\0~~∼. You must end your strings with the null character or else you won't be able to perform operations like printf.

`string.h` library

`strcat`

`strncat`

`strcmp` ~~~c

~~~

`strncmp` ~~~c

~~~

Section 1: Pointers

Any programming language at its very heart is about storing and moving information around. In fact, to your computer, there is no difference between an `int` or `char` or even `char*` etc., they are simply a sequence of bits that are passed around in your machine.

It is the human component and only this component that makes the decision that an `int` expresseses our idea of quantity or number and `char` expresses our idea of characters. If you were to "ask" your computer what is stored at a particular memory address, the response will always be "I don't know. Who cares?". The first byte at this location could be interpreted as a `char` or the first 4 or 8 bytes at the same location (depending on your machine) could be interpreted as an `int`, `char*` or any other data type.

We can let the compiler know we've decided to switch our interpretation of the data we stored in a variable. It's called casting

```c
int main(int argc, char ** argv) {

    char mem[] = "Hello World";
    // creating a int pointer to the array allows us to use the same
    // data, but interpreted differently.
    int * p = (int*) &(mem[0]);

    printf("Size of Integer: %d bytes\n", sizeof(int));
    printf("%d", *p);
}
```

```
$ gcc casting.c
$ ./a.out
Size of Integer: 4 bytes
1819043144
```

> Note: Since the size of integer is only 4 bytes, not all of "Hello World" has been translated to a integer value. only the first 4 bytes.

The point being, **memory isn't specifically "formatted" to store your requested data type**. An `int` is stored in the same way as a `char`,`char*`, `double` etc. - a sequence of bytes. A data type simply defines how many bytes we should read starting from a location.

Pointers in C are like any other datatype. When declaring a pointer, your computer simply allocates 8 bytes in memory to store a sequence of 8 bytes. It is us who decided to first interpret these 8 bytes as a number then as a memory address.

Whenever you pass in a variable to a function in C, it is always copied. If its an `int`, the bytes of size `sizeof(int)` are copied. If its a `char*` the bytes of size `sizeof(char*)` are copied. These copied values exist as local variables inside your function, specifically parameters, and go out of scope when the function returns. The same logic is true for returning a value from a function, the value is copied and sent back to the caller.

Passing in a pointer into a function has **absolutely no difference**, the value the pointer holds (i.e the 8 bytes that store the memory address) are copied and stored as a local variable in the function (i.e your parameter). Failure to distinguish the above ideas about copy on pass and copy on return leads to main source of confusion about pointers. People tend to think that pointers are passed to function in a special way - **they are not**. They are done in the exact same way an `int` or `char` etc. would be passed in. It's just that one has to realize its not what the pointer is pointing to that is passed, but the value of the pointer. (the memory address it holds)

Common Mistakes and Bugs

Uninitialized Pointer - Dereferenced before assignment

```
int main(int argc, char ** argv) {

    int * p;

    *p = 3; // BAD! p doesn't point to a valid location location in memory.
            // Attempting to write will trigger SIGSEGV
}
```

Dangling Pointer - Access

```
int main(int argc, char ** argv) {

    char * p = malloc(20); // allocate 20 bytes
    strcpy(p, "hello");
    ...
    free(p);
    strcat(p, "world"); // BAD! memory was already freed
                        // Attempting to access will trigger SIGSEGV
}
```

Memory Leak

```
int main(int argc, char ** argv) {

    char * p = malloc(20); // allocate 20 bytes
    strcpy(p, "hello");

    char a[] = "Cake";

    p = &(a[0]); // BAD! p was pointing to memory allocated on heap
                 // reassignment of p now removes all references to that
                 // memory location. cannot free.
}
```

Dereferencing a pointer and overwriting the value

Section 2: Linked Lists

You will use linked lists a lot during your upper year courses. It is a very fundamental data structure and mastery in manipulating linked lists is vital.

A linked list comes with some basic operations as you have seen in lecture. While even a lackluster CS student could see how linked lists could be created in Python with a class containing a `next` property. The pointers in C actually give us a whole new level of control. The ability of being able to create a pointer and dereference it somewhere later in your code is a very powerful tool.

Let's review some of the basic operations:

Let's first define our Node `struct` ~~~c typedef struct **node** Node; // don't worry about this, this is called // prototyping. It's so we can use the // typedef keyword `Node` in our Node struct // before we've actually defined it.

typedef struct **node** { int v; Node * next } Node; ~~~

`INSERT(head,k)`

Note the double pointer to head allows us to change the head node pointer by dereferencing.

```
/**
 * Inserts the Node k into the beginning linked list head.
 */
void insert (Node** head, Node* k) {

    if (!head) return;
    k->next = *head;
```

```
    *head = k;
}

/**
 * Inserts the Node k into the end of linked list head.
 */
void insert (Node** head, Node* k) {

    if (!head) return;

    Node * curr = *head;

    // if this is the first node in the list, overwrite head.
    if (!curr) *head = k; return;

    // find last node in list
    while (curr->next != NULL) curr = curr->next;
    // set the next to our new node.
    curr->next = k;
}
```

DELETE(head,v)

Delete is slightly more complicated as we have to consider the edge cases where the node is the head of the list and the end of the list

```
/**
 * Deletes Node with value v from linked list head
 * Returns 0 if node with value v is not found, 1 otherwise
 */
int delete (Node** head, int v) {

    if (!head) return 0;

    Node * curr = *head;
    Node * prev = NULL;

    // look for the node with value v.
    while (curr->v != v) {
        // keep track of previous node, we will need it later
        prev = curr;
        curr = curr->next;
        // if we reach end of list prematurely, just return
        if (!curr) return 0;
    }
    // it helps to consider the exit condition of this loop
    // exit condition: negation of while loop condition
    // -> curr->v == v
    // this means the loop will exit with curr->v == v

    if (!prev) { // if the node we want to delete is the head node.

        *head = curr->next; // overwrite the head node pointer to the next one.

    } else { // otherwise it occurs at the end of the list or the middle

        // if it's in the middle, this sets the previous node pointer
        // to the one after curr (curr being the one we want to delete).
        // if it's at the end, curr->next == NULL, hence still correct
        // as the prev node will now have prev->next = NULL.
        // (i.e the new end  of the list)
        prev->next = curr->next;
    }
    // free up the memory for curr (the deleted node)
    free(curr)
    return 1;
}
```

Exercises

1. Write a function `void llreverse(Node** head)`, that reverses the linked list rooted at `head`.

2. ______________________________

Questions, Comments and Suggestions can be directed to me at david.yue@mail.utoronto.ca

We are also your CSC A48 Computer Science Representatives for AMACSS, feel free to ask course material questions through this email as well :)

Requests to see this study guide in a different format are welcome :)